CAPES de mathématiques
Option Informatique–Session 2018

Le sujet est constitué de deux problèmes indépendants.

Problème n° 1 : suite de Lucas

Rappels et notations. Pour x un nombre réel, il existe un plus grand entier inférieur ou égal à x, appelé *plancher* de x ou *partie entière* de x. Ce nombre entier est noté $\lfloor x \rfloor$. Il existe un plus petit entier supérieur ou égal à x, appelé *plafond* de x.

Les fonctions `ceil` et `floor` du module `math` de Python calculent respectivement le plafond et le plancher d'un nombre flottant x : Ainsi `ceil(1.24)` vaut 2 et `floor(1.24)` vaut 1.

Soit a un nombre réel strictement positif. On désigne par $\log_a$ la fonction logarithme en base a : si x est un nombre réel strictement positif,

$$\log_a x = \frac{\ln x}{\ln a},$$

où ln désigne la fonction logarithme néperien.

Dans ce problème, on étudie plusieurs algorithmes de calcul des nombres de Lucas.

Ces nombres sont les termes de la suite $(L_n)_{n\geqslant 0}$ définie par les relations suivantes :

$$\begin{cases} L_0 = 2, \\ L_1 = 1, \\ L_n = L_{n-1} + L_{n-2} \text{ pour } n \geqslant 2. \end{cases}$$

I. Calculer L_2, L_3, L_4, L_5, L_6 et L_7.

$L_2 = 3$, $L_3 = 4$, $L_4 = 7$, $L_5 = 11$, $L_6 = 18$, $L_7 = 29$.

II. Montrer que pour tout $n \geqslant 0$, L_n est un entier naturel.

Récurrence simple laissée au lecteur.

III. On considère l'équation $x^2 - x - 1 = 0$.

1. Montrer que cette équation possède deux solutions, l'une positive que l'on note ϕ, l'autre négative que l'on note $\hat{\phi}$.

L'équation du second degré a pour discriminant 5, strictement positif, elle admet donc deux solutions :

$$\phi = \frac{1+\sqrt{5}}{2} > 0 \text{ et } \hat{\phi} = \frac{1-\sqrt{5}}{2} < 0.$$

Des valeurs approchées à 10^{-4} près de ces deux nombres sont $\phi \approx 1{,}6180$ et $\hat{\phi} \approx -0{,}6180$.

2. Justifier que

$$\phi + 1 = \phi^2, \qquad \hat{\phi} + 1 = \hat{\phi}^2, \qquad \phi + \hat{\phi} = 1, \qquad \phi\hat{\phi} = -1.$$

ϕ et $\hat{\phi}$ sont solutions donc $\phi + 1 = \phi^2$ et $\hat{\phi} + 1 = \hat{\phi}^2$.
La somme des solutions est $\phi + \hat{\phi} = 1$ et leur produit $\phi\hat{\phi} = -1$.

IV. Montrer que pour tout entier naturel n, $L_n = \phi^n + \hat{\phi}^n$. On pourra raisonner par récurrence.

On vérifie l'égalité $L_n = \phi^n + \hat{\phi}^n$ pour $n = 0$ et $n = 1$. Soit $n \geqslant 2$, on suppose l'égalité vérifiée jusqu'au rang $n - 1$.
Alors $L_n = L_{n-1} + L_{n-2} = \phi^{n-2}(1 + \phi) + \hat{\phi}^{n-2}(1 + \hat{\phi}) = \phi^n + \hat{\phi}^n$.
La récurrence s'enclenche bien.

V. On donne cette valeur approchée du logarithme en base 10 de ϕ :

$$\log_{10} \phi = \frac{\ln \phi}{\ln 10} \approx 0{,}2090.$$

Montrer que pour tout entier naturel p,

$$n \geqslant 5p \Longrightarrow L_n \geqslant 10^p.$$

Comme $\log_{10} \phi = 0{,}209 \pm 10^{-4}$, on en déduit que $\phi^5 > 10$. En outre $|\hat{\phi}| < 1$, donc, pour tout entier naturel n, $\left|\hat{\phi}^n\right| < 1$.

Ainsi, pour $n \geqslant 5p$, $L_n \geqslant \phi^n - 1 \geqslant \phi^{5p} - 1 > 10^p - 1$. Comme L_n est clairement un nombre entier, $L_n \geqslant 10^p$.

VI. **1.** On propose la fonction suivante pour calculer le n-ième nombre de Lucas.
On rappelle que `**` est l'opérateur Python d'élévation à la puissance.

```
0 from math import *
1
2 def lucas1(n):
3     if n == 0:
4         return 2
5     phi = (1+sqrt(5))/2
6     phi2 = (1-sqrt(5))/2
7     return phi**n + phi2**n
```

L'évaluation de `[lucas1(n) for n in range(8)]` renvoie la liste

```
[2, 1.0, 3.0, 4.0, 7.000000000000001, 11.000000000000002,
        18.000000000000004, 29.000000000000007].
```

Pourquoi ne s'agit-il pas d'une liste d'entiers ?

`phi` et `phi2` sont des flottants, donc le résultat renvoyé par `lucas1` est également un flottant. Les décimales non nulles sont dues au fait que le calcul avec des flottants est un calcul approché.

2. On propose maintenant la fonction suivante pour calculer le n-ième nombre de Lucas.

```
0 from math import *
1
2 def lucas2(n):
3     if n == 0:
4         return 2
5     phi = (1+sqrt(5))/2
6     if n%2 == 0:
7         return ceil(phi**n)
8     else:
9         return floor(phi**n)
```

L'évaluation de `[lucas2(n) for n in range(8)]` renvoie la liste

`[2, 1, 3, 4, 7, 11, 18, 29]`.

Expliquer le choix de ces fonctions en lignes 6 à 9 et démontrer que, si les calculs en flottants sont exacts, `lucas2(n)` calcule bien L_n.

On a déjà dit que $\left|\hat{\phi}^n\right| < 1$ et que $\hat{\phi}$ est négatif.

Si n est pair, $L_n = \phi^n + \hat{\phi}^n$ est dans l'intervalle $[\phi^n, 1 + \phi^n[$ et c'est donc bien le plafond de ϕ^n.
Si n est impair, L_n est dans l'intervalle $]-1 + \phi^n, \phi^n]$ et c'est donc bien le plancher de ϕ^n.
Si les calculs sont exacts, `lucas2` renvoie donc bien le bon résultat.

3. Un calcul exact montre que $L_{36} = 33385282$, mais `lucas2(36)` renvoie la valeur 33385283. Comment expliquez-vous cela ?

En fait, les calculs effectués sur les nombres flottants sont approchés. Et la valeur exacte de ϕ^{36} devrait être strictement inférieure à 33385282 (mais de très peu), alors que python calcule une valeur approchée un tout petit peu plus grande : cette approximation explique l'erreur, puisqu'elle provoque une erreur d'une unité sur le calcul de `floor(phi**n)`.
(Effectivement $\phi^{36} \approx 33385281{,}99999997004\ldots$)

VII. On souhaite écrire une nouvelle fonction de calcul des termes de la suite de Lucas. Pour éviter tout problème lié au calcul avec des flottants, on souhaite ne travailler qu'avec des entiers, sur lesquels Python calcule de manière exacte.

1. Recopier et compléter la fonction suivante, qui renvoie la valeur de L_n.

```
0 def lucas3(n):
1     if n == 0:
2         return 2
3     if n == 1:
4         return 1
5     a,b = (2,1)
6     for i in range(n):
7         a,b = (..., ...)
8     return ...
```

Corrigé :

```
0 def lucas3(n):
1     if n == 0:
2         return 2
3     if n == 1:
4         return 1
5     a,b = (2,1)
6     for i in range(n):
7         a,b = (b,a+b)
8     return a
```

2. Déterminer un invariant de boucle qui précise, en fonction de la valeur de `i`, les valeurs des variables `a` et `b` avant et après l'exécution de la ligne 7 et en déduire que `lucas3` renvoie un résultat exact.

Montrons qu'on a toujours $a = L_i$ et $b = L_{i+1}$ avant l'exécution de la ligne 7.
En effet c'est vrai la première fois, quand $i = 0$, et si c'est vrai pour un i fixé, au tour suivant, on aura bien $a = L_{i+1}$ et $b = L_i + L_{i+1} = L_{i+2}$.
On en déduit qu'on a toujours $a = L_{i+1}$ et $b = L_{i+2}$ à la fin de l'exécution de la ligne 7. En particulier, lors du dernier passage dans la boucle, $i = n - 1$ et donc $a = L_n$ et $b = L_{n+1}$.
Il faut donc bien que `lucas3` renvoie la valeur de $a = L_n$ à la fin de l'appel.

3. Pour $n \geqslant 2$, combien d'additions sont effectuées par `lucas3(n)` ?

`lucas3` effectue une addition par passage dans la boucle. Au total, elle effectue donc n additions (si $n \geqslant 2$).

VIII. 1. Démontrer que, pour tout entier $n \geqslant 1$,

$$L_n^2 - L_{n+1}L_{n-1} = 5\,(-1)^n.$$

Pour $n = 1$, on a bien $L_1^2 - L_2L_0 = 1 - 2 \times 3 = -5 = 5\,(-1)^1$.
Supposons l'égalité vraie jusqu'au rang $n \geqslant 1$, et montrons qu'elle reste vraie au rang $n + 1$.
On dispose de :

$$L_{n+1}^2 - L_{n+2}L_n = L_{n+1}^2 - (L_n + L_{n+1})L_n = L_{n+1}(L_{n+1} - L_n) - L_n^2 = L_{n+1}L_{n-1} - L_n^2 = -5\,(-1)^n =$$

ce qui est le résultat attendu.
Autre solution : on développe $L_n^2 - L_{n+1}L_{n-1} = (\phi^n + \hat{\phi}^n)^2 - (\phi^{n+1} + \hat{\phi}^{n+1})(\phi^{n-1} + \hat{\phi}^{n-1})$ en tenant compte que $\phi\hat{\phi} = -1$.

2. Démontrer que, pour tout entier $n \geqslant 1$,

$$\begin{cases} L_{2n} &= L_n^2 - 2(-1)^n, \\ L_{2n+1} &= L_nL_{n+1} - (-1)^n. \end{cases}$$

Pour $n = 1$, on a bien $L_2 = 3 = 1^2 - 2(-1) = L_1^2 - 2(-1)^1$ et $L_3 = 4 = 1 \times 3 - (-1) = L_1L_2 - (-1)^1$.
Supposons les deux égalités vraies jusqu'au rang $n - 1$ (où $n \geqslant 2$) : $L_{2n-2} = L_{n-1}^2 + 2(-1)^n$ et $L_{2n-1} = L_{n-1}L_n + (-1)^n$. En sommant, on obtient :

$$L_{2n} = L_{n-1}(L_{n-1} + L_n) + 3(-1)^n = L_{n-1}L_{n+1} + 3(-1)^n,$$

et, en utilisant le résultat obtenu en 5.a, on en déduit $L_{2n} = L_n^2 - 5(-1)^n + 3(-1)^n = L_n^2 - 2(-1)^n$.

La première égalité est donc vérifiée au rang n. Montrons qu'il en est de même pour la seconde.

On écrit $L_{2n+1} = L_{2n-1} + L_{2n} = L_{n-1}L_n + (-1)^n + L_n^2 - 2(-1)^n = L_n(L_{n-1} + L_n) - (-1)^n = L_nL_{n+1} - (-1)^n$, ce qui conclut la récurrence.

Autre solution : on développe $L_n^2 - 2(-1)^n = (\phi^n + \hat{\phi}^n)^2 - 2(-1)^n = \phi^{2n} + \hat{\phi}^{2n} + 2(-1)^n - 2(-1)^n = L_{2n}$.

Puis $L_nL_{n+1} - (-1)^n = (\phi^n + \hat{\phi}^n)(\phi^{n+1} + \hat{\phi}^{n+1}) - (-1)^n = \phi^{2n+1} + \hat{\phi}^{2n+1} + (\phi\hat{\phi})^n(\phi + \hat{\phi}) - (-1)^n = L_{2n+1}$.

IX. 1. Si k est un entier, que calcule l'expression Python suivante : `1 - 2*(k % 2)` ?

Si k est pair, `1 - 2*(k % 2)` vaut $1 = (-1)^k$. Si k est impair, `1 - 2*(k % 2)` vaut $1 - 2 = -1 = (-1)^k$. Dans tous les cas on trouve $(-1)^k$.

On rappelle que l'opérateur `%` calcule le reste dans la division entière : `7 % 3` vaut 1 ; `8 % 3` vaut 2 et `9 % 3` vaut 0.

2. Recopier et compléter la fonction récursive suivante, de sorte que `lucas4(n)` renvoie le couple (L_n, L_{n+1}).

```
0  def lucas4(n):
1      if n == 0:
2          return (2,1)
3      if n == 1:
4          return (1,3)
5      k = n // 2
6      u = 1 - 2*(k % 2)
7      a,b = lucas4(...)
8      if n % 2 == 0:
9          return (..., ...)
10     else:
11         return (..., ...)
```

Attention au cas où n est impair, où on écrit $n = 2k + 1$, $a = L_k$, $b = L_{k+1}$ puis $L_n = L_{2k+1} = L_kL_{k+1} - (-1)^k$ et $L_{n+1} = L_{2k+2} = L_{k+1}^2 - 2(-1)^{k+1} = L_{k+1}^2 + 2(-1)^k$.

```
def lucas4(n):
    if n == 0:
        return (2,1)
    if n == 1:
        return (1,3)
    k = n // 2
    u = 1 - 2*(k % 2)
    a,b = lucas4(k)
    if n % 2 == 0:
        return (a*a - 2*u, a*b - u)
```

```
    else:
        return (a*b - u, b*b + 2*u)
```

3. Pour tout entier $n \geqslant 2$, exprimer en fonction de n le nombre d'appels récursifs que réalise `lucas4(n)`.

Soit $\alpha(n)$ le nombre d'appels récursifs effectués par `lucas4(n)`. On a $\alpha(0) = \alpha(1) = 0$ et, pour $n \geqslant 2$, $\alpha(n) = 1 + \alpha(\lfloor \frac{n}{2} \rfloor)$. On en déduit que $\alpha(n) = \max\{p \in \mathbb{N}, 2^p \leqslant n\} = \lfloor \log_2 n \rfloor$ où $\log_2$ désigne le logarithme en base 2.
On est passé d'un temps de calcul linéaire (avec `lucas3`) à un temps de calcul logarithmique.

X. Pour n un entier naturel, on considère le vecteur colonne de $\mathbb{R}^2$ défini par

$$V_n = \begin{pmatrix} L_n \\ L_{n+1} \end{pmatrix}.$$

On considère également la matrice $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Pour tout entier $n \geqslant 1$, exprimer V_n en fonction de A et de V_{n-1}, puis exprimer V_n en fonction de A, de n et de V_0.

On a $V_n = AV_{n-1}$ donc, par une récurrence immédiate, $V_n = A^n V_0$.

On représente dans la suite une matrice carrée $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ par la liste de deux listes

`[[a,b], [c,d]]`.

XI. 1. écrire en Python une fonction `prodMat(M1,M2)` qui prend en arguments deux matrices 2×2 représentées par des listes comme indiqué ci-dessus, et qui renvoie la liste qui représente la matrice produit $M1 \times M2$.

Corrigé :

```
def prodMat(M1,M2):
    a1,b1 = M1[0][0],M1[0][1]
    c1,d1 = M1[1][0],M1[1][1]
    a2,b2 = M2[0][0],M2[0][1]
    c2,d2 = M2[1][0],M2[1][1]
    return [ [a1*a2+b1*c2, a1*b2+b1*d2],
                  [c1*a2+d1*c2,c1*b2+d1*d2] ]
```

2. Combien l'appel `prodMat(M1,M2)` effectue-t-il d'additions ? de multiplications ?

On dénombre 4 additions et 8 multiplications.

3. On considère la fonction (naïve) d'élévation d'une matrice à une puissance entière suivante.

```
0 def puissanceMat(M,p):
1     R = [[1,0],[0,1]]
2     for i in range(p):
3         R = prodMat(R,M)
4     return R
```

Combien l'appel `puissanceMat(A,p)` réalise-t-il d'appels à `prodMat` en fonction de p ?

Bien sûr cela fera p appels à `prodMat`.

XII. L'algorithme d'exponentiation rapide repose sur la remarque que

$$a^{2p+1} = (a^p)^2 a = b \times b \times a,$$

où $b = a^p$, quand $a^{2p} = b \times b$. Selon la parité de p, il faut donc 1 ou 2 multiplications de plus pour calculer a^{2p+1} ou a^{2p} connaissant a^p. Ainsi, pour calculer a^{122}, par exemple :

$$\begin{aligned} a^{122} &= (a^{61})^2 \\ &= (a.(a^{30})^2)^2 \\ &= (a.((a^{15})^2)^2)^2 \\ &= (a.((a.(a^7)^2)^2)^2)^2 \\ &= (a.((a.(a.(a^3)^2)^2)^2)^2)^2 \\ &= (a.((a.(a.(a.a^2)^2)^2)^2)^2)^2. \end{aligned}$$

Ainsi, on se contente de 10 multiplications.

1. On propose la fonction suivante qui implémente l'exponentiation rapide pour les matrices 2×2.

```
0 def puissanceMatRapide(M,n):
1     R = [[1,0], [0,1]]
2     P = M
3     while n > 0:
4         if n%2 == 1:
5             R = prodMat(R,P)
6         P = prodMat(P,P)
7         n = n // 2
8     return R
```

Montrer que par cette méthode, le nombre d'appel à la fonction `prodMat` est majoré par $1 + 2\lfloor \log_2 p \rfloor$.

On passe exactement $1 + \lfloor \log_2 n \rfloor$ dans la boucle des lignes 3 à 7. Chaque passage donne lieu à 1 ou 2 appels à `prodMat` selon que n est pair ou impair. Au total on peut donc majorer le nombre d'appels par $2 + 2\lfloor \log_2 n \rfloor$.

2. Exprimer en fonction de M et de i la valeur de la matrice P, après la ième itération de la boucle `while`.

À chaque itération, on élève au carré la matrice P. Donc $P = M^{2^i}$.

3. Soit $n = \overline{c_p \dots c_0}$ l'écriture de n en base 2. Montrer que, pour tout entier i compris entre 1 et p, la valeur de l'entier n après la ième itération de la boucle `while` est donée par

$$n = \sum_{j=i}^{p} c_j 2^{j-i}.$$

Après la première itération, on a divisé n par 2 : donc $n = \overline{c_p \dots c_1} = \sum_{j=1}^{p} c_j 2^{j-1}$.

Supposons qu'à la fin de la i-ème itération on ait $n = \overline{c_p \dots c_{i+1} c_i} = \sum_{j=i}^{p} c_j 2^{j-i}$.

À l'itération suivante on aura divisé n par 2, donc on aura $n = \overline{c_p \dots c_{i+1}} = \sum_{j=i+1}^{p} c_j 2^{j-i-1}$, ce qui enclenche la récurrence.

4. Montrer que pour tout entier i compris entre 1 et p, la valeur de la matrice R après la ième itération de la boucle `while` est donnée par $R = M^k$, avec

$$k = \sum_{j=0}^{i} c_j 2^j.$$

On procède par récurrence sur n dont l'écriture binaire s'écrit $n = \overline{c_p \dots c_1 c_0}$.
Si n est pair ($c_0 = 0$), après la première itération on a toujours $R = I_2 = M^0$ et $0 = c_0 2^0$; si n est impair ($c_0 = 1$), après la première itération on a $R = M = M^1$ et $1 = c_0 2^0$.
Donc dans les deux cas, après la première itération, $R = M^k$ avec $k = \sum j = 0^0 c_j 2^j$.

Supposons qu'après la i-ème itération on ait $R = M^k$ avec $k = \sum_{j=0}^{i-1} c_j 2^j$.

Rappelons qu'après la i-ème itération $n = \overline{c_p \dots c_{i+1} c_i}$ et $P = M^{2^i}$.
Si $c_i = 0$, à l'issue de la $(i+1)$-ème itération, on aura $R = M^k$ avec la même valeur de $k = \sum_{j=0}^{i-1} c_j 2^j = \sum_{j=0}^{i} c_j 2^j$ puisque $c_i = 0$.

Si $c_i = 1$, à l'issue de la $(i+1)$)ème itération, on aura $R = M^k \times M^{2^i} = M^{k'}$ avec $k' = k + 2^i = \sum_{j=0}^{i} c_j 2^j$.

La récurrence s'enclenche bien.

5. Exprimer le nombre d'exécutions de la boucle `while` en fonction de n puis démontrer la correction de l'algorithme proposé, c'est-à-dire que `puissanceMatRapide(M,n)` calcule effectivement la puissance désirée.

On a déjà dit qu'il y a $1 + \lfloor \log_2 n \rfloor = 1 + p$ passages dans la boucle, où on a écrit $n = \overline{c_p \dots c_0}$.

À l'issue de la dernière itération, on a montré que $R = M^k$ avec $k = \sum_{j=0}^{1+p-1} c_j 2^j = n$.

C'est ce qu'on voulait !

XII. Proposer le code d'une fonction `lucas5(n)` qui retourne la valeur du nombre de Lucas L_n en utilisant la fonction `puissanceMatRapide`.

Corrigé :

```
def lucas5(n):
    A = [[0,1],[1,1]]
    R = puissanceMatRapide(A,n)
    return 2*R[0][0]+R[0][1]
```

Problème n° 2 : emplois du temps et graphes d'intervalles

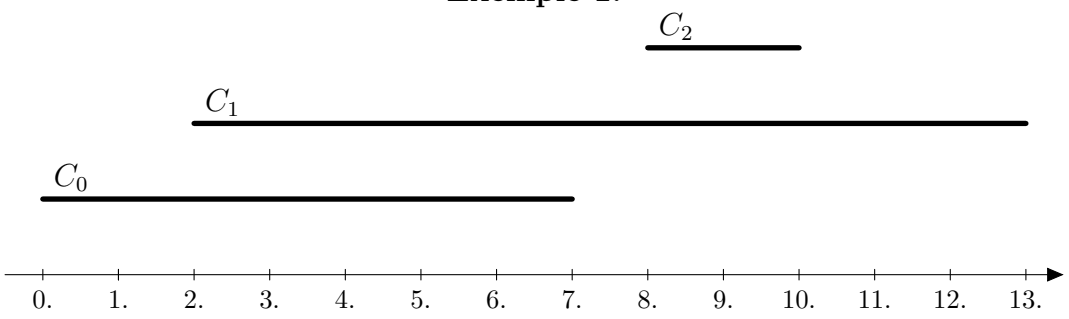
Dans ce problème, on s'intéresse à l'allocation des salles d'un lycée à partir des horaires des cours.

La donnée du problème est une liste de cours. Un cours est simplement représenté par un intervalle $[\texttt{deb}, \texttt{fin}[$, où `deb` est l'instant de début du cours et `fin` est l'instant de fin du cours. Les instants peuvent être décomptés en heures ou en minutes au fil de la journée selon les besoins ; dans ce sujet, les instants sont des nombres entiers et l'unité de temps n'est pas précisée.

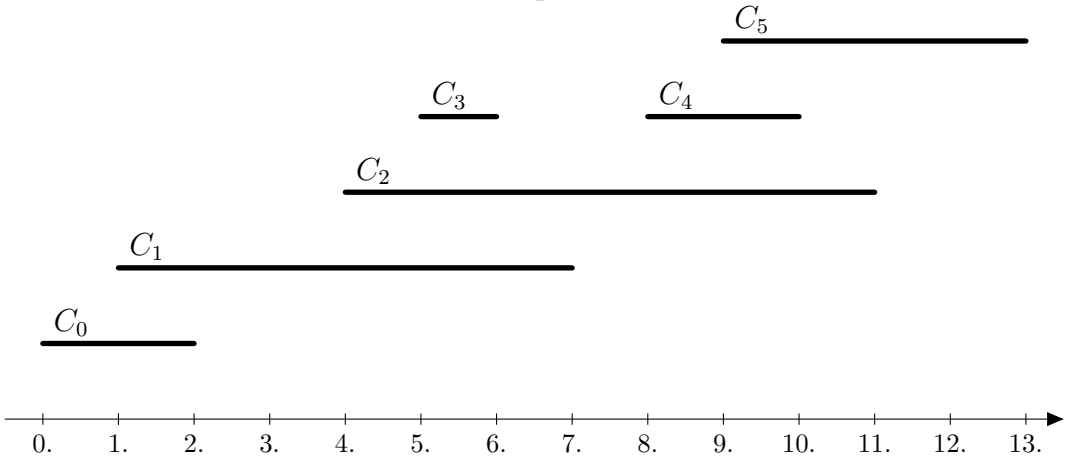
À chaque cours, on doit allouer une salle. Deux cours peuvent se voir allouer la même salle, uniquement si leurs intervalles sont disjoints. En particuliers, deux cours représentés par les intervalles $[4, 6[$ et $[6, 8[$ peuvent se voir allouer la même salle. L'objectif est d'utiliser un minimum de salles.

On présente ci-dessous, deux instances de ce problème.

Exemple 1.



Exemple 2.



Dans l'exemple 1, le cours C_2 est représenté par l'intervalle $[8, 10]$.

I. Pour l'allocation des salles, on parcourt simplement les cours par instants de début croissants et on alloue systématiquement la salle disponible avec le numéro le plus petit.

1. Décrire, sans justification, les allocations ainsi obtenues pour chacun des exemples précédents.

   Pour l'exemple 1, on utilise deux salles : la première pour les cours C_0 et C_2, la seconde pour le cours C_1.
   Pour l'exemple 2, on utilise trois salles : la première pour les cours C_0 et C_2, la seconde pour les cours C_1 et C_4, la troisième pour les cours C_3 et C_5.

2. Déterminer le nombre minimal de salles nécessaires pour l'allocation dans les deux exemples précédents.

   Pour l'exemple 1, on ne peut faire moins que deux salles, les deux cours C_0 et C_1 étant incompatibles.
   Pour l'exemple 2, C_1, C_2 et C_3 sont deux à deux incompatibles : il faut utiliser au moins 3 salles.

On admet pour la suite que le nombre optimal de salles nécessaire pour une liste de cours est le nombre maximal d'intervalles s'intersectant mutuellement en un instant donné.

II. on s'intéresse dans cette question à une première modélisation du problème. Le cours représenté par la liste Python `[d,f]` se déroule sur l'intervalle mathématique $[d, f[$.

1. Déterminer les listes d'intervalles représentant les exemples 1 et 2.

   Pour l'exemple 1 : `[[0,7],[2,13],[8,10]]`,
   pour l'exemple 2 : `[[0,2],[1,7],[4,11],[5,6],[8,10],[9,13]]`.

2. Compléter la définition suivante :

```
0 def insere(l,elt):
```

   où `l` est une liste d'entiers triée dans l'ordre croissant et `elt` est un entier, et qui renvoie une liste triée obtenue par insertion à sa place de `elt` dans `l`. On notera que la liste `l` peut être vide.

   Corrigé :

```
0 def insere(l,elt):
1     n = len(l)
2     i = 0
3     while i<n and l[i]<elt:
4         i = i + 1
5     return l[:i]+[elt]+l[i:]
```

On suppose pour la suite que l'on dispose d'une fonction similaire `insereBis` qui opère sur des listes de listes plutôt que sur des listes d'entiers, le critère de tri étant l'ordre des premiers éléments de chaque sous-liste. Plus précisément, la fonction `insereBis` a pour en-t ête `def insereBis(LL,li):` et a pour effet d'insérer une liste `li` à la place adéquate dans la liste de listes `LL`.

III. Pour automatiser l'allocation des salles, on va utiliser une autre modélisation. L'intervalle [`deb`, `fin`[représentant le cours numéro i va être représenté par deux événements, modélisés par des listes de longueur 3 : `[deb,i,0]` et `[fin,i,1]`. Une liste d'intervalles pourra ainsi être représentée par une liste d'événements, c'est-à -dire une liste de listes de longueur 3 de la forme `[instant,num,0 ou 1]`.

1. Compléter la définition

```
def traduit(liste_intervalles):
```

qui prend en argument une liste d'intervalles représentés par des listes de longueur 2 et qui renvoie une liste d'événements (listes de longueur 3) correspondante. Notons que pour n intervalles, on obtient $2n$ événements.

Corrigé :

```
def traduit(liste_intervalles):
    liste_evt = []
    for i in range(len(liste_intervalles)):
        cours = liste_intervalles[i]
        liste_evt.append([cours[0],i,0])
        liste_evt.append([cours[1],i,1])
    return liste_evt
```

2. Pour l'efficacité des algorithmes de résolution, on va travailler sur une liste d'événements triée par instants croissants. On appellera *agenda* toute liste d'événements triés par instants croissants. Quels sont les agendas correspondant aux exemples 1 et 2 ?

L'agenda de l'exemple 1 est `[[0,0,0],[2,1,0],[7,0,1],[8,2,0],[10,2,1],[13,1,1]]`.

Exemple 2 : `[[0,0,0],[1,1,0],[2,0,1],[4,2,0],[5,3,0],[6,3,1],[7,1,1],[8,4,0],[9,5,0],[10,4,1],[11,2,1],[13,5,1]]`.

3. Compléter la définition

```
def agenda(liste$\_$evt):
```

qui prend en argument une liste d'événements (listes de longueur 3) obtenue par un appel à la fonction traduit et qui renvoie l'agenda correspondant. On pourra utiliser la fonctions `insereBis`. Déterminer la complexité de la fonction `agenda` pour une liste de $2n$ événements.

On réalise un tri par insertion.

```
def agenda(liste_evt):
    LL = []
    for evt in liste_evt:
        LL = insereBis(LL,evt)
    return LL
```

IV. On a demandé à des élèves d'écrire une fonction qui vérifie qu'une liste d'événements donnée contient bien autant de fin que de début, et dans le bon ordre, mais sans tester l'appariemment cours par cours, c'est-à -dire sans considérer les numéros d'intervalles.

1. Parmi les quatres solutions proposées, déterminer (sans justifier) la ou les réponses correctes et préciser leur complexité en fonction de la longueur de la liste en paramètre.

```
def valideA(agenda):
    c = 0
    for e in agenda:
        if e[2] == 0: c += 1
        else: c -= 1
        if c < 0: return False
        else : return True

def valideB(agenda):
    n,c,i,b = len(agenda),0,0,True
    while b and (i < n):
        if agenda[i][2] == 0: c += 1
        else: c -= 1
        i += 1
        b = (c >= 0)
    return c == 0

def valideC(agenda):
    for i in range(len(agenda)):
        if agenda[i][2] == 0:
            b = False
            for j in range(i+1,len(agenda)):
                if agenda[j][2] == 1: b = True
            if not b : return(b)
    return(True)

def valideD(agenda):
    c = 0
    for e in agenda:
        c += 1 - 2*e[2]
        if c < 0: return False
    return c == 0
```

`valideA` est incorrecte à cause du `return True`.

`valideB` est correcte et de complexité linéaire.

`valideC` est incorrecte : un seul événement fin peut terminer tous les débuts.

`valideD` est correcte et de complexité linéaire.

2. Adapter une des fonctions précédentes pour écrire une fonction `intersection_max` qui calcule le nombre maximal d'intervalles qui s'intersectent mutuellement. Justifier la correction de l'approche.

On réutilise le code de `valideD` en observant que `c` compte le nombre d'intervalles ouverts mais pas encore fermés. Il s'agit de déterminer la valeur maximale atteinte par ce compteur.

```
def intersection_max(agenda):
    c = 0
    cmax = 0
    for e in agenda:
        c += 1 - 2*e[2]
```

```
        if c>cmax:
            cmax = c
    return cmax
```

3. En utilisant les fonctions précédentes, écrire une fonction `nbr_optimal` qui à partir d'une liste d'intervalles (modélisation initiale), calcule le nombre de salles nécessaire. Quelle est la complexité de la fonction `nbr_optimal` en fonction du nombre d'intervalles ?

Il s'agit simplement de combiner les fonctions précédentes. La complexité reste linéaire.

```
def nbr_optimal(li):
    return intersection_max(agenda(traduit(li)))
```

V. On utilise à chaque instant une liste de booléens pour indiquer si une salle est disponible ou non.

1. On a demandé à un élève d'écrire une fonction qui étant donné une liste de booléens, calcule le plus petit entier i tel que la case d'indice i vaille `True`.

La fonction renverra `-1` si un tel indice n'existe pas. Corriger sa fonction et en préciser la complexité en fonction de la longueur de la liste passée en paramètre.

```
def plus_petit_vrai(liste):
       n = len(liste)
       while liste[i] and (i < n) : i+= 1
       if i = n: return -1
      else: return i
```

Il faut initialiser `i`. Le bon test est `not(liste[i])` et on doit d'abord vérifier `i<n`. D'autre part la comparaison entre `i` et `n` doit se faire avec `==`.

La complexité est évidemment linéaire.

```
def plus_petit_vrai(liste):
    n = len(liste)
    i = 0
    while i<n and not(liste[i]):
        i += 1
    if i == n: return -1
    else: return i
```

2. En utilisant les fonctions précédentes, compléter la fonction suivante qui étant donnée une liste d'intervalles (listes de longueur 2), calcule une liste d'allocations des salles, toujours en allouant la salle disponible avec le plus petit numéro. Dans cette liste, la case d'indice le numéro du cours contient le numéro de la salle allouée.

```
def allocation(liste_intervalles):
    nb_cours = ...
    liste = ...
    nb_salles = ...
    salles_dispos = [True]*nb_salles
    alloc = [-1]*nb_cours
    for l in liste:
```

```
        if l[2] == 0 :
            alloc[l[1]] = ...
            salles_dispos[...] = False
        else :
            salles_dispos[...] = ...
    return(alloc)
```

Corrigé :

```
def allocation(liste_intervalles):
    nb_cours = len(liste_intervalles)
    liste = agenda(traduit(liste_intervalles))
    nb_salles = nbr_optimal(liste_intervalles)
    salles_dispos = [True]*nb_salles
    alloc = [-1]*nb_cours
    for l in liste:
        if l[2] == 0 :
            alloc[l[1]] = plus_petit_vrai(salles_dispos)
            salles_dispos[alloc[l[1]]] = False
        else :
            salles_dispos[alloc[l[1]]] = True
    return(alloc)
```